Funzione main() e prima sintassi c++.

```
int main()
{
    int x;
    std::cout<<"Benvenuto c++"<<std::endl;
    system("pause"); //è una funzione che esegue comandi dos. pause è un comando dos per blocca
                     //l'esecuzione del programma
    std::cout << "Dammi un intero!" <<std::endl;
    std::cin >> x;
    std::cout << x;
    return 0;
}
```

Se in java avevamo la classe main, in c++ abbiamo la funzione main(). La funzione main restituisce un intero, che ha valore 0 se il main ha funzionato in maniera corretta, o ha valore -1 se la funzione non ha funzionato in maniera corretta.

La direttiva **#include** consente di importare le librerie esterne (in pratica fa la stessa cosa di “import” in java). In questo caso la direttiva #include mi permette di importare la libreria **iostream**, che contiene classi in grado di acquisire dati da input e classi in grado di generare dati in output. Le direttive devono essere scritte furoi dal main. In particolare la classe **cin** mi permette di eseguire le operazioni di input, mentre la classe **cout** mi permette di eseguire le operazioni di output. Esempio:

std::cout<<”Dammi un intero!”<<endl;

nota1: l’oggetto cout e l’**operatore di invio <<** (che è sempre fornito dalla libreria iostream) mi permettono di stampare sullo schermo la stringa “Dammi un intero”. Ma per utilizzare l’oggetto cout di libreria iostream devo specificare dove la libreria è

contenuta. Questo lo faccio con l’**operatore di scope** :: (è equivalente all’operatore punto in java). La libreria iostream è contenuta nel namespace **std**.
Per scrivere un’altra volta sullo schermo devo usare sempre l’operatore di invio <<. In questo caso, con **endl**(equivale a \n in java) voglio specificare un ritorno a capo sullo schermo.

Nota2: l’istruzione std::cout<<”Dammi un intero!”; e equivalente all’istruzione System.out.print(“Dammi un intero!”); in java. Da notare che in java usavo il metodo print() per stampare su riga, invece in c++ uso l’operatore di invio <<.

Nota3: l’oggetto cout può stampare su una linea dati di tipo diverso sfruttando l’operazione di concatenazione attraverso l’operatore di invio <<. Esempio:

```
.
.
int x=3;
cout<<”il numero ”<<x;
.
.
std::cin >> x;
```

nota1: l’operatore di invio **>>** è puntato verso la variabile x, questo significa che il valore che andrò a prendere da input (cioè da tastiera) sarà memorizzato nella variabile x.

nota2: l’istruzione std::cin>>x; e equivalente alle istruzioni in java:

```
Scanner nome_Oggetto= new Scanner(System.in);
x=nome_Oggetto.nextInt();
```

nota3: in c++, l’oggetto che è incaricato di prendere il valore di input non deve essere creato per essere utilizzato. Invece in java, uno dei oggetti incaricati di prendere da input un dato, deve essere

creato prima di essere utilizzato; per esempio l'oggetto di classe System.in deve essere creato prima di essere utilizzato;

nota4: sfruttando l'operatore di invio >> posso anche memorizzare due valori in due variabili diverse con una singola istruzione:

```
.
.
int a;
int b;
cin>>a>>b;
.
.
```

nota5: in c++ gli oggetti non vengono utilizzati attraverso i loro rispettivi metodi, ma vengono utilizzati attraverso gli operatori (per esempio, gli operatori di invio << e >>) della loro rispettiva libreria(per esempio, iostream);

nota5: la funzione system("pause") può anche non essere messa, perché viene eseguita implicitamente dal compilatore;

nota6: ogni volta che utilizzo un oggetto della libreria iostream, devo specificare dove la libreria iostream è contenuta, cioè nel name space std. Per evitare questa cosa posso usare la clausola **using** specificando il nome del namespace, ovvero std:

```
#include <iostream>
using namespace std;

int main()
{
    int x;
   cout<<"Benvenuto c++"<<endl;
    system("pause"); //è una funzione che esegue comandi dos. pause è un comando dos per blocca
                     //l'esecuzione del programma
    cout << "Dammi un intero!" <<endl;
    cin >> x;
    cout << x;
    return 0;
}
```

Creazione di una classe.

```
#include <string>
using namespace  std;
class Persona
{
private:
    //string nome,cognome;
    string nome;
    string cognome;
    int eta;

public:
    Persona(string nome, string cognome, int eta)
    {
        this->nome = nome;
        this->cognome = cognome;
        this->eta = eta;
    }
    string getNome() {
        return nome;
    }
    void setNome(string nome) {
        this->nome = nome;
```

Il **nome della classe** deve avere la prima lettera maiuscola (come in java). Gli **attributi** di una classe in c++ gli dichiaro privati. Gli

attributi gli posso scrivere in concatenazione (questo in java non lo posso fare. Devo specificare per ogni attributo che è privato).
Il **tipo string** è scritto con la lettera minuscola (in java deve essere scritto in lettera maiuscola). Il tipo string non è primitivo quindi per utilizzarlo lo devo importare con la direttiva #include string; (in java anche se non è primitivo non lo devo importare). Ogni qualvolta dichiaro una variabile string devo anche specificare da quale **namespace** la classe string proviene, ovvero std:

std:: string nome_variabile;

Per evitare questa cosa devo utilizzare la parola chiave **using** specificando il nome del namespace:
using namespace std;

nota1: tutte le librerie che importo faranno parte del namespace std. Quindi deve dichiarare il namespace std solo una volta;

nota2: dichiarando il namespace std nella classe Persona non lo dovrò dichiarare anche nella funzione main() dato che la classe Persona è stata importata dalla funzione main(). Inoltre la funzione main() importando la classe Persone ha importato in automatico anche le librerie che la classe Persona ha importato (quindi anche la libreria string).

Come in java, il costruttore e i metodi per accedere agli attributi privati devono essere dichiarati pubblici. Quando ho una sequenza di metodi pubblici posso dichiarare un public generico e mettere i metodi in cascata(come abbiamo visto nel esempio sopra).

Nota1: in c++ la parentesi graffa che chiude il blocco di una classe deve essere usata in coppia con un punto e virgola: };

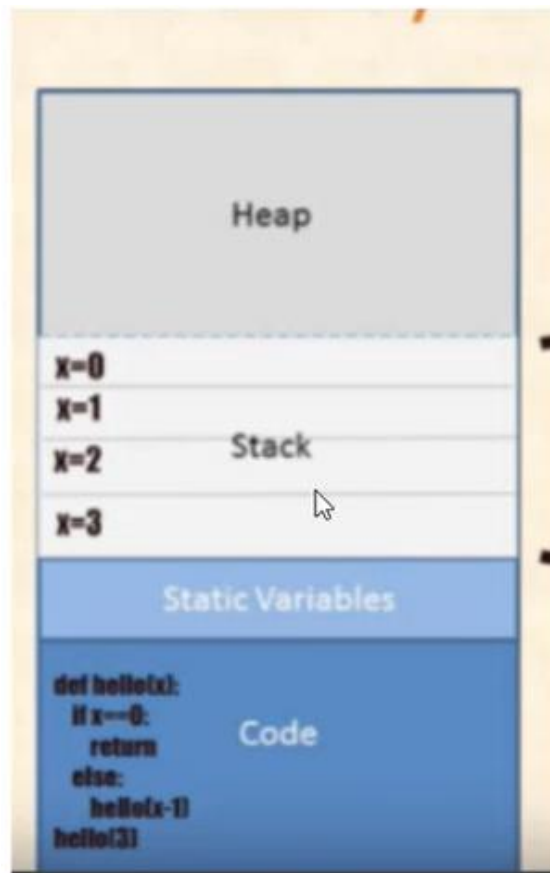
nota2: quando aggiungiamo una classe al progetto visual studio mi crea due file uno con estensione .h (dove è presente il codice della classe) e uno con estensione.cpp (dove è presente la direttiva #include al file con estensione.h). In pratica, quando definiamo una classe la dobbiamo includere nella funzione main(). In particolare, è il file della classe con estensione .cpp ad dover essere incluso nella funzione main() con la direttiva include:

#include “Nome_Classe.cpp”;

nota3: quando includo una classe alla funzione main(), con la direttiva #include, devo mettere il nome della classe tra virgolette (invece il nome di una libreria è messo nella notazione a diamante <>);

nota4: un progetto deve avere solo un file che definisce la funzione main(). Quindi per segnare effettivamente il file con funzione main(), come il primo file che deve essere compilato ed eseguito, devo cliccare con il tasto destro sul file e selezionare “imposta come progetto di avvio”.

Allocazione statica.

Un **processo** (programma in esecuzione) occupa un segmento della memoria RAM. Questo spazio di memoria generalmente è suddiviso in quattro segmenti di memoria:

- Heap: contiene variabili allocate dinamicamente;
- Stack: contiene la pila dei record di attivazione. Ogni volta che una funzione viene chiamata viene creato un record di attivazione;
- Static Variables: contiene le costanti e le variabili globali;
- Code: contiene le istruzioni del programma in linguaggio macchina.

Ora vediamo con un esempio come viene allocata in maniera statica la memoria:

```
1  #include <iostream>
2  using namespace std;
3  //COSTANTI E VARIABILI GLOBALI
4  int a;
5  //DEFINIZIONE FUNZIONI
6  int raddoppia(int x) {
7     int b;
8     b=2*x;
9     return b;
10 }
11
12 int main() {
13    int y;
14    cout<<"Inserisci un intero: ";
15    cin>>y;
16    y=raddoppia(y);
17    cout<<"Complimenti hai raddoppiato: "<<y<<endl;
18 }
```

Nota1: la variabile intera “a”, dato che è globale viene allocata nell’**area globale**. Il tempo di vita di una variabile globale è equivalente alla durata dell’intero processo.

Nota2: Lo **stack** è utilizzato per memorizzare innanzitutto il record di attivazione della funzione main costituito dalla variabile y, sul quale viene impilato il record di attivazione della funzione raddoppia() in corrispondenza della chiamata che avviene nella riga 16. Questo record di attivazione è costituito da:

- l’indirizzo di rientro nella funzione main (indirizzo della funzione), ossia l’indirizzo che nell’area del programma individua l’istruzione successiva a quella che ha invocato la funzione raddoppia().
- il parametro passato alla funzione raddoppia() (parametro locale).

- la variabile locale b della funzione raddoppia() (che è anche il valore di ritorno).

In particolare, nell'esempio il secondo record di attivazione, relativo alla funzione raddoppia, viene eliminato dallo stack quando la funzione stessa termina e l'esecuzione del programma riprende nel sottoprogramma chiamante, il main, nella riga 17. In questo modo, quando si rientra nel main la variabile del parametro passato alla funzione e la variabile locale b vengono distrutte.

Nota3: è importante osservare che l'uso dello stack consente ad una funzione di invocare un'altra funzione e, come abbiamo visto nell'esempio di prima, ciò determina la creazione di un nuovo record di attivazione in cima alla pila dei record di attivazione dello stack. Più in generale, il meccanismo della pila dei record di attivazione permette il corretto flusso di esecuzione del programma e di gestione dei tempi di vita delle variabili locali, quando all'interno di un programma si hanno le chiamate alle funzioni. Questo meccanismo permette anche che una funzione possa invocare al suo interno se stessa (meccanismo detto di ricorsione).

Puntatori.

Un **puntatore** è un oggetto il cui valore rappresenta l'indirizzo di un altro oggetto o di una funzione, o di un dato semplice. In altre parole un puntatore è una variabile che non indica un valore ma un indirizzo di memoria:

```
#include <iostream>
using namespace std;
/*
  sintassi della dichiarazione di un puntatore in c++
  tipo* nome_variabile;
  */

int main()
{
    int x = 3;
    // &x indica l'indirizzo di memoria della variabile x
    int* ptr_x = &x;//ptr_x non è una copia della variabile x
    cout << "indirizzo della variabile x " << ptr_x << endl;
    cout << "indirizzo " << &x<<endl;

    //prendere il valore di x attraverso il suo puntatore ptr_x
    cout << "Valore di x attraverso il suo puntatore " << *ptr_x << endl;
    x = 10;
    cout << "Valore di x attraverso il suo puntatore " << *ptr_x << endl;

}
```

Nota1: la dichiarazione di un puntatore inizia specificandone prima il tipo, poi bisogna mettere prima del nome del puntatore l’operatore *(infatti l’operatore * indica che la variabile è usato come un puntatore). Esempio:

int *ptr_x;

nota2: per ottenere l’indirizzo dove ho memorizzato un numero intero uso l’operatore &, il cui risultato può essere assegnato ad un puntatore. Esempio:

ptr_x=&x;

nota3: per accedere all’oggetto riferito da un puntatore è usato l’operatore *:

cout<< “Valore di x attraverso il suo puntatore: “ << *ptr_x<<endl;

nota4: in questo caso abbiamo usato l'operatore * per accedere al valore di un intero e non ad un oggetto.

Passaggio di parametri per valore e per riferimento.

```
#include <iostream>
using namespace std;

void swapRef(int*, int*);
int main()
{
    int x, y;
    cout << "x=";
    cin >> x;

    cout << "y=";
    cin >> y;


    swapRef(&x, &y);//corretta.scambia la x e la y
    cout << "x=" << x << " " << "y=" << y << endl;

}

//passaggio di parametri per riferimento
//chiamata per riferimento. alla funzione vengono passati i puntatori (indirizzi) dei
parametri
void swapRef(int* x, int* y)
{
    int temp = *x;
    *x = *y;
    *y = temp;

}
```

La funzione **swapRef()** ha lo scopo di scambiare il valore di due dati. Quindi la funzione swapRef() deve essere chiamata per riferimento (solo una funzione chiamata per riferimento può modificare il valore dei suoi argomenti). Infatti, gli argomenti della funzione swapRef() non sono valori di "x" e "y" ma sono gli indirizzi in memoria di "x" e "y" che sono stati presi utilizzando l'operatore &. Questi indirizzi saranno inviati come parametri alla funione swapRef(). La funzione riceve questi indirizzi e gli memorizza nei rispettivi parametri formali, che in una funzione chiamata per riferimento saranno sempre puntatori(in c++ solo i puntatori possono contenere indirizzi, e una variabile diventa un puntatore utilizzando l'operatore *).

Nota1: abbiamo messo il prototipo della funzione swapRef() all’inizio del programma (cioè sopra la funzione main). Così, il compilatore effettua il controllo fra numero di parametri passati e numero di parametri indicati nel prototipo. In questo modo, il compilatore non compila un codice che potrebbe essere sbagliato.
Programmare in C – Funzioni: i prototipi di funzioni | Elettronica Open Source (emcelettronica.com)

Nota2: il **prototipo** di una funzione è la dichiarazione della funzione stessa. Quest’ultima è composta dal tipo di ritorno della funzione, dal nome della funzione e tra parentesi il tipo di parametri nell’ordine in cui compaiono nella chiamata alla funzione.

Nota3: l’operatore* mi trasforma una variabile in un puntatore. Successivamente per accedere alla memoria associata al puntatore devo sempre utilizzare l’operatore *:

```
int *a;

*a=3;
```

Nota4: in java non ci sono puntatori, ma variabili riferimento. E per accedere o modificare i valori degli elementi riferiti dalle variabili riferimento utilizzo opportuni metodi.

I reference.

Deferenzazione: ogni volta che voglio accedere alla cella di memoria alla quale il puntatore è associato, devo usare sul puntatore l'operatore *).
Per trasformare una variabili in un reference devo usare l'operatore &. L'inizializzazione di un reference deve avvenire subito dopo la sua dichiarazione. Inoltre per inizializzare un reference devo usare l'operatore = come viene fatto per inizializzare una variabile semplice (solo che un reference contiene un indirizzo). I reference una volta inizializzati non possono essere più sovrascritti (a differenza dei puntatori), cioè allo stesso reference non posso associare un nuovo oggetto.
I **reference**, come i puntatori sono contenitori di indirizzo, ma non è necessario deferenziarli per accedere alla cella di memoria (i reference vengono usati come se fossero normale variabili).
Esempio:

```
int var1=5;
int var2=6;
int var3=7;
int* ptr_var1;
ptr_var1=&var1;
int& ref_var2=var2;
cout<<*ptr_var1;
cout<<ref_var2;
ptr_var1=&var3;
cout<<*ptr_var1;
```

nota1: un reference può essere inizializzato anche da un puntatore, ma deve essere deferenziato (visto che l'inizializzazione di un reference funziona come quella di una normale variabile):

```
int var=8;
int* ptr= &var;
int& ref=*ptr;
```

nota2: nell’ambito della chiamata di una funzione, il meccanismo dei reference nasconde il fatto che ad essere passato come parametro è un indirizzo e non una copia. Infatti l’utilizzo dei puntatori rende esplicito il passaggio di un indirizzo (per utilizzare l’indirizzo a cui il puntatore è associato devo utilizzare l’operatore &).

Dichiarazione e inizializzazione di un array e allocazione statica della memoria.

Al momento della dichiarazione di un array in c++ è possibile solo postporre le parentesi quadre al nome del array (in java invece è possibile sia anteporre sia postporre le parentesi quadre al nome del array). Inoltre in c++, quando un array non deve essere un valore di ritorno delle funzione dove è stato creato, allora l’allocazione della memoria al momento della creazione del array è statica. Sempre in c++, un array è trattato come un puntatore (una variabile viene contrasegnata come puntatore applicando l’operatore *). Infatti l’inizializzazione di un array è associata all’indirizzo del primo elemento del array:

```
tipo* nomeArray []; // dichiarazione array
nomeArray={1,2,3};
```

Dichiarazione e inizializzazione di un array e allocazione dinamica della memoria.

Quando vogliamo che il valore di ritorno di una funzione è un array, allora dobbiamo usare l’operatore **new** per creare l’array. Questo permette di allocare la memoria in modo dinamico. In pratica salvando l’array nel heap e non nello stack permette ai dati contenuti nel array di non essere cancellati quando la funzione (dove è stato dichiarato e inizializzato l’array) restituisce il controllo alla funzione che l’ha chiamata:

```
int* creaArray()
{
int* arr= new int[] {1, 2, 3};

return arr;
}
```

nota1: l’array in c++ viene trattato come un puntatore. Infatti il puntatore che rappresenta un array e associato all’indirizzo del primo elemento del array;

nota2: in c++, agli elementi di un array posso accedere o attraverso il puntatore (che è un accesso più veloce) e sfruttando l’aritmetica dei puntatori, o attraverso l’accesso classico agli elementi del array (come in java):

```
#include <iostream>
using namespace std;

int main()
{
int* arr=creaArray();

for(int i=0; i<3; i++)
{
//cout<<*(arr+i)<<” “;
//cout<<arr[i]<<” “;
}
return 0;
}
```

nota3: se la dimensione del array fosse stata data da input allora avrei dovuto trovare un modo per determinare la dimensione del array.

L'operatore delete.

L'**operatore delete** ha la funzione di deallocare l'area di memoria relativa all' heap, e ha la seguente sintassi:

delete nome_Puntatore;

In altre parole l'operatore delete non cancella il puntatore nome_Puntatore o il suo contenuto, ma libera l'area di memoria associata al puntatore, rendendola disponibile per ulteriori allocazioni.
L'operatore delete rappresenta l'unico mezzo per deallocare oggetti memorizzati nella memoria heap, che rimangono memorizzati fino alla fine del programma anche quando il loro riferimento viene perso, esempio:

```
int* punt = new int; // alloca un int nell'area heap
int a;
punt = &a; // assegna a punt un indirizzo dell'area stack, per cui
                // l'oggetto int dell'area heap non è più raggiungibile
               // quindi l'unico modo per liberare la memoria associata
              // associata all'oggetto int è utilizzare l'operatore delete
```

nota1: in java la gestione della memoria heap avviene in automatico attraverso il Garbage collection che cancella gli oggetti che non hanno più un riferimento.

Le classi.

Nota1: in c++, quando stiamo definendo una classe e abbiamo importato la libreria string se la vogliamo utilizzare bisogna specificare il namespce std utilizzando la parola chiave using, fuori dalla dichiarazione della classe. Inoltre se voglio utilizzare la libreria string nella funzione main() non la devo importare dato che ho importato la classe dove ho utilizzato la libreria string.

Nota2: come in java anche in c++ utilizzo l'operatore new per creare un oggetto. In particolare l'operatore new mi consente di memorizzare l'oggetto nella memoria heap. Inoltre la variabile che contiene un oggetto deve essere dello stesso tipo della sua classe. In più in c++ questa variabile deve essere trasformata in un puntatore:

```
// la  classe ha il costruttore vuoto
NomeClasse *nomePuntatore=new NomeClasse();
```

Nota3: inoltre posso associare allo stesso oggetto più puntatori:

```
NomeClasse *nomePuntatore1=new NomeClasse();
NomeClasse *nomePuntatore2=nomePuntatore1;
```

nota4:nomePuntatore1 non è la copia di nomePuntatore2, ma sono due puntatori associati allo stesso oggetto. Ogni modifica che faccio su nomePuntatore2 inciderà anche su nomePuntatore1, poiché tutti e due sono associati allo stesso oggetto.

Nota5: l'operatore * è usato su un puntatore per prendere il dato associato al puntatore stesso, che sia esso un dato semplice oppure un oggetto. In particolare, quando voglio invocare il metodo di un oggetto mi serve l'oggetto stesso:

```
*nomePuntatore.nomeMetodo();
```

Il puntatore this.

Il **puntatore this** è usato solo in una classe e non in una funzione. Quando il puntatore this è usato nel costruttore della classe è associato all'oggetto che ha invocato il costruttore (i costruttori permettono di inizializzare gli oggetti che gli hanno invocati). Così il puntatore this evita di generare uguaglianze tra attributi dell'oggetto e parametri del costruttore. Un puntatore this non deve essere dichiarato, la sua dichiarazione è implicita nella classe. Il puntatore this è usato anche da altri metodi, ma è sempre associato all'oggetto che ha invocato il metodo stesso.
In realtà, il puntatore this è usato in una classe in combinazione con altri elementi:

```
class Persona
{
private:
string nome;
string cognome;
int eta;

public persona(string nome, string cognome, int eta)
{
this - > nome = nome; // è equivalente a (*this).nome=nome;
this - > cognome = cognome;
this - > eta = eta;
}

};
```

nota: la notazione - > potevo anche utilizzarla nella funzione main() quando vado ad invocare il metodo di un oggetto:
.

.

```
Persona *p=new(nome, cognome, eta);
cout<<p- > toStirng()<<endl; // è equivalente a (*p).toString()
```

Overloading di una funzione.

```
string toString()
{
return
“nome”+nome+”cognome”+congnome+”età”+to_string(eta);
}
```

Dato che il metodo toString() restituisce solo una stringa e l’attributo eta è un intero io non posso fare la concatenazione come ho fatto con nome e cognome. Per risolvere la questione devo utilizzare la funzione to_string() sull’attributo eta (in pratica faccio un overloading della funzione to_string()). Più semplicemente potevo dichiarare eta come una stringa e non come numero.
to_string - C++ Reference (cplusplus.com)
Inoltre, l’operatore di concatenazione + concatena solo una volta una stringa ad un numero. Quindi devo usare un’altra volta la funzione to_string() per lavorare solo sulle stringhe.

Metodo costruttore copia (overloading del costruttore della classe).

In una classe posso definire un metodo costruttore copia, che prende come parametro la copia del puntatore del oggetto che voglio copiare. Grazie alla copia del puntatore accedo agli attributi del oggetto per cui è associato. Quindi posso copiare l’oggetto; esempio:
//classe Persona
.
.

```
Persone(Persona *p)
{
nome=p - >getNome();
cognome=p - >getCognome();
eta=p - >getEta();
}
.
.
//funzione main()
.
.
Persona*  pCopy= new Persona(p);
.
.
```

Nota: p è il puntatore associato all’oggetto p che ho creato prima. Il puntatore lo utilizzerò nel costrutore di pCopy per accedere agli attributi dell’oggetto p.

Copia di un oggetto e allocazione statica della memoria.

```
//funzione main()
.
.
Persona p(“Mario”, “Rossi”, 24);
setNam(p);
setNam(&p);
setNamRef(&p);
.
.

//classe Persona
.
.
public persona(string nome, string cognome, int eta)
{
}
.
.
void setNamCop(Persona p)
{
p.setNome(“Luca”);
}

void setNamPunt(Persona * p)
```

```
{
p - >setName(“Marco”);
}

void setNamRef(Puntatore &p)
{
p.serName();
}
.
.
```

Nota1: quando creo un oggetto nello stack(allocazione statica) e non nello heap(allocazione dinamica) la variabile che conterà l’oggetto non deve essere un puntatore ma una semplice variabile del tipo della classe del oggetto. Quindi userò la stessa variabile per invocare un metodo, o come argomento dell’invocazione di un metodo (o funzione). Infatti la funzione setNam() viene invocato per valore e non per riferimento, quindi riceverà come parametro la copia dell’oggetto p. Ogni modifica fata alla copia dell’oggetto p non inciderà sull’oggetto originale p. Mentre la funzione setNam() è invocato per riferimento infatti riceve come parametro la copia del puntatore di p. Quindi ogni modifica fatta alla copia del puntatore di p inciderà sull’oggetto p, dato che il puntatore di p e la sua copia puntato allo stesso oggetto. Inoltre la funzione setNameRef() è sempre invocato per riferimento ma il suo argomento è un reference. Il parametro che arriverà a setNameRef() sarà la copia del reference p (i reference in c++ sono variabili semplici e non puntatori. Quindi non hanno bisogno del operatore * per accedere al oggetto che associano).

Nota2:Per oggetto creato staticamente non dobbiamo utilizzare la notazione - > per invocare un metodo, ma utilizzeremo l’operatore punto.
Nota3: se non utilizzo l’operatore new gli oggetti vengono creati in modo statico e non dinamio.
Nota4: in java non distinguiamo puntatori e reference. Ci sono solamente variabili associabili ad oggetti.
Nota5: in java tutti gli oggetti creati vengono salvati nel heap.
Nota6: la funzione seNam è in overloading.

La differenza tra allocazione dinamica e allocazione statica della memoria.

Si mette in evidenza che le variabili globali e le variabili locali (quelle cioè dichiarate all'interno di una funzione) in C/C++ devono essere sempre **allocate staticamente**, cioè la loro dimensione in byte deve essere conosciuta a priori dal compilatore. In generale, non è noto a priori quanti record di attivazione saranno allocati nello stack durante l'esecuzione del programma, ma il compilatore sarà in grado di calcolare la dimensione di un record di attivazione dall'analisi del codice sorgente. Questo rende possibile una gestione corretta dello stack.

L'area di **memoria dinamica**, denominata **heap**, consente di allocare spazi di memoria la cui dimensione è nota solo a runtime (in fase di esecuzione del programma), e per questo è detta memoria dinamica. Una variabile viene allocata nell'heap utilizzando opportune istruzioni messe a disposizione dal linguaggio di programmazione. Una variabile allocata dinamicamente ha la caratteristica di avere un tempo di vita che non è legato a quello delle funzioni. In particolare, non è legata al tempo di esecuzione della funzione in cui essa viene creata, nel senso che sopravvive anche dopo che la la funzione termina. Ciò consente di creare una variabile dinamica in una funzione, di utilizzarla in altre funzioni e di distruggerla (deallocarla esplicitamente con le istruzioni opportune) in una funzione ancora diversa.

Esempio di errore tipico che può succedere quando scambiamo un'allocazione dinamica per un'allocazione statica:

```
int main() {
   int n;
   cout << "Quanti sono gli elementi del vettore? << endl;
   cin >> n;
   //ATTENZIONE QUESTO MODO DI OPERARE E' SBAGLIATO!!!
   int v[n]; //Allocazione statica errata!!!
   ....
}
```

Introduzione all’allocazione dinamica della memoria in C++ – LabSquare.it

Ereditarietà.

Nota1: dato che visual studio quando creo una classe mi crea anche un file con estensione .cpp, per evitare che per ogni sottoclasse ci sia anche il suo relativo file con estensione .cpp, quando creo una sottoclasse devo specificare che il file con estensione .cpp sia quello della sopraclasse e che la classe base sia ovviamente la sopraclasse. Quando creo una classe A, che non deriva dalla sovraclasse, e le quali istanze saranno utilizzate in una sottoclasse, devo specificare come file di estensione .cpp quello della sovraclasse e come classe base il nome della sopraclasse.
Nota2: in visual basic una classe viene creata andando nel menu sopra su progetto e cliccando su aggiungi classe.

Nota3: visusl basic quando mi crea una sottoclasse mi include in automatico anche la sopraclasse. Ma questa istruzione la posso cancellare visto che la sopraclasse è stata inclusa nel file con estensione .cpp che ho specificato al momento della creazione della sottoclasse.

Nota4: fuori dalla funzione main() devo includere il file della sopraclasse con estensione .cpp

Una sottoclasse in c++ viene dichiarata in questo modo:

class NomeSottoclasse:
  public NomeSopraclasse

In c++ viene usata una speciale sintassi (liste di distribuzione) per specificare che la prima istruzione che deve essere eseguita dal costruttore di una sottoclasse e il richiamo del costruttore della sopraclasse:

public:
NomeCostruttoreSottoclasse():NomeCostruttoreSopraclasse()
{
.
.
}

Nota: gli attributi della sopraclasse gli devo dichiarare protected, perché così sono accessibili direttamente dagli oggetti delle sottoclassi. Se invece avessi dichiarato private gli attributi della sopraclasse,essa sarebbe stata incapsulata. In altre parole gli attributi privati della sopraclasse non sarebbero stati accessibili direttamente (anche dalle sottoclassi) ma solo tramite i metodi pubblici ereditati dalle sottoclassi stesse.

Esempio di ereditarietà:

```
#pragma once
#include <string>
using namespace  std;

class Persona
       {
       protected:

              string nome;
              string cognome;
              int eta;

       public:
              Persona(string nome, string cognome, int eta)
              {
                     this->nome = nome;
                     this->cognome = cognome;
```

```
            this->eta = eta;
        }
        Persona(string nome, string cognome) :Persona(nome, cognome, 30) {}

        string getNome() {
            return this ->nome;
        }
        void setNome(string nome) {
            this->nome = nome;
        }

    // devo usare il puntatore this?

      string toString() {
            return this->nome + " " + this->cognome + " " + to_string(eta);
     }



    };
```

Nota1: il secondo costruttore della classe Persona è in l'overloading rispetto al primo. Inoltre, la prima istruzione del secondo costruttore della classe Persona è scritta usando le liste di distribuzione. Invece il corpo del costruttore è vuoto.

Nota2: metto in overloading la funzione to_string() dandogli come argomento l'attributo intero eta. Così facendo la funzione to_String() restituisce una stringa.

```
#pragma once

class Studente :

public Persona
{
//private come default
  string matricola;

public:
    Studente(string nome, string cognome, int eta, string
matricola):Persona(nome,cognome,eta) {
        this->matricola = matricola;


   }
    Studente(string nome, string cognome, string matricola) :Persona(nome, cognome) {
        this->matricola = matricola;
        // this - > nome;
        // this - > cognome;


    }

    string toString() {
        return Persona::toString() + " " + this ->matricola;
    }
```

```
};
```

Nota1: nella classe Studenti in tutte e due i costruttori ho utilizzato le liste di distribuzione per specificare che la prima istruzione che deve essere eseguita è l’invocazione di uno dei costruttori della sopraclasse (in java, per invocare il costruttore della sopraclasse uso la parola chiave super seguita dall’operatore punto, invece in c++ viene usato il nome della sopraclasse seguto dal carattere di escape ::).

Nota2: il metodo toString() ereditato dalla sottoclasse Studente è in overriding. Per evitare di scrivere gli attributi nome, cognome e eta invoco il metodo toString() della sopraclasse Persona utilizzando il carattere di escape (in java per invocare un metodo della sopraclasse nella sottoclasse è usata la parola chiave super seguita dall’operatore punto).

Nota3: in una sottoclasse quando un metodo ereditato è in overriding non posso più accedere alle operazioni che definiva prima, perché sono state riscritte. In realtà, le operazioni originarie di un metodo in overriding sono ancora disponibili nella sopraclasse, devo solo invocare il metodo attraverso la sopraclasse(anziché attraverso la sottoclasse) tramite i carattere di escape.

Programmazione orientata agli oggetti in linguaggio C++ (edutecnica.it)

Array dinamici in c++.

Nota: in java, la dichiarazione di un array dinamico comincia specificando la classe Vector (la classe Vector è quella che contiene i metodi utili alla manipolazione di un array dinamico) usando la notazione a diamante per specificare il tipo degli oggetti che formano l'array dinamico, infine specificare il nome dell'array dinamico:

```
Vector <Nome_classe> nome_Array_Dinamico;
```

Siccome in c++ un oggetto non è associato ad una variabile riferimento ma ad un puntatore, al momento della dichiarazione di un array dinamico nella notazione a diamante ci sarà un puntatore, specificare il tipo di oggetti che formeranno un array dinamico. Inoltre, per specificare che lo stesso array dinamico è un puntatore, metto fuori dalla notazione a diamante l'operatore *:

```
vector <Nome_Classe*>* nome_Array_Dinamico;
```

Nota1: in c++ un array dinamico può essere formato anche da dati semplici come gli interi (in java invece un array dinamico è formato solo da oggetti).

Un array dinamico è allocata dinamicamente. Quindi per creare un array dinamico utilizzerò l'operatore new:

```
nome_Array_Dinamico=new vector<Nome_Classe*>;
```

Nota1: ovviamente per utilizzare la classe vector la dobbiamo importare.

Nota2: le librerie che non abbiamo definito non vanno specificate con la prima lettera maiuscola. Infatti la classe vector ha la prima lettera minuscola.

In java, per inizializzare un array dinamico utilizzavamo il metodo addElement(). Invece in c++ possiamo dichiarare, allocare e inizializzare con una singola istruzione un array dinamico:

```
vector<Studente*>* arrayDinamico=new vector<Studente*>{new Studente(“Mario”, “Rossi”, “a001”), new Studente(“Luca”, “Bianchi”, “a001”};
```

Nota1: la dimensione dell’array dinamico è determinato in base a quanti elementi metto nelle parentesi graffe.

Inoltre alcuni elementi di un array dinamico di una classe non vector gli posso inserire in un: array dinamico della classe vector:

```
Studente* arr1[]= {new Studente(“Mario”, “Rossi”, “a001”), new Studente(“Luca”, “Bianchi”, “a001”};
vector<Studente*>* arr2=new vector<Studente*>(arr,1 arr1+0);
```

Nota1: dopo che ho dichiarato e creato un array dinamico, posso specificare esplicitamente la sua dimensione mettendola tra parentesi tonde.
Nota2: la dimensione di un array dinamico la determino sfruttando l’aritmetica dei puntatori.
Nota3: dato che ho sommato 0 al puntatore arr1, la dimensione di arr2 è di un elemento.

Nota4: il metodo at() della classe vector prende l’elemento del array dinamico nella posizione specifica tra parentesi tonde:

```
cout<<arr2 - >at(0)- >getNome()<<endl;
```

Nota5: arr2 è costituita da puntatori, oltre ad essere esso un puntatore. Gli oggetti rappresentati da puntatori possono invocare i propri metodi utilizzando l’operatore - >. Quindi utilizzo l’operatore - > invocare il metodo at() della classe vector, e per invocare il metodo getNome() della classe Persona.

Nota6: per stabilire la dimensione di un array dinamico in maniera generale utilizzo l’operatore sizeof():

```
Studente* arr1[]={new Studente(“Mario”, “Rossi”, “a001”), new Studente(“Mario”, “Rossi”, “a002”)};
vector<Studente*>* arr3=new vector<Studente>(arr1, arr1+sizeof(arr1)/sizeof(Studente*));
```

Nota7: la dimensione di un array dinamico rappresenta in pratica quanta memoria devo allocare. Nell’esempio il puntatore arr rappresenta la prima cella di memoria da allocare, e la somma tra il puntatore arr1 e il numero di elementi contenuti nell’array(rappresentato dall’espressione sizeof(arr)/sizeof(Studente*)) rappresenta l’ultima cella da allocare.

Nota8: questo modo di calcolare la dimensione di un arr1 in modo generale è fondamentale perché gli elementi (che dobbiamo inserire in arr3) fossero stati inizializzati da input, la dimensione di arr1 non sarebbe nota a priori, e quindi non saprei come calcolare la dimensione di arr3 senza l’espressione sizeof(arr1)/sizeof(Studente*).

La parola chiave static in c++.

Dichiarando un attributo static lo rendo comune a tutti gli oggetti che andrò a creare. Questo significa che un attributo static non è legato alla creazione di un oggetto. In particolare, non è possibile inizializzare un attributo static tramite la lista di inizializzazione del costruttore della classe. In c++, inizializzo l’attributo statico nel file .cpp che è stato creato assieme al file .h(che contiene la definizione della classe). In pratica nel file .cpp, non dichiaro esplicitamente che l’attributo è statico, ma dichiaro che è un attributo di classe e non di istanza (cioè del oggetto della classe) utilizzando il carattere scope (::). Infine, sempre nel file .cpp, inizializzo l’attributo statico:

tipoAttributo NomeClasse::nomeAttributo=0;

Nota: quando ci sono sottoclassi, l’attributo statico lo inizializzo nel file.cpp della sovraclasse. Inoltre il file .cpp della sovraclasse contiene sia i file .h delle sottoclassi sia i file .h delle altre classi.

Nel file .h (dove ho definito la classe) dichiaro un’altra volta l’attributo statico, questa volta esplicitamente, però senza inizializzarlo:

static tipoAttributo nomeAttributo;

Nota1: dichiaro un attributo statico per non occupare inutilmente memoria. Per esempio, se devo creare un attributo che tiene conto del numero di

oggetti che ho creato allora sarebbe inutile che lo crei ogni volta che creo un oggetto, perciò dichiaro l'attributo statico.
Nota2: dopo che un attributo statico è stato inizializzato può essere acceduto da tutti i metodi della classe (costruttore compreso) per operazioni di lettura o scrittura.
Nota3: se devo definire un metodo che accede solamente ad un attributo statico o modifica solamente un attributo statico, allora devo dichiarare questo metodo statico a sua volta.
Nota4: metodi statici possono accedere solo a variabili statiche, e possono invocare solo altri metodi statici. Ma se un metodo statico riceve come parametro un puntatore associato ad un oggetto di una classe, allora il metodo statico può accedere agli attributi di quel'oggetto.
Nota5: ascoltare tempo 01-42-50.

Polimorfismo per dati.

Metodi virtuali.

Quando utilizziamo il polimorfismo per dati in c++ e vogliamo invocare un metodo in overriding (metodo ereditato da una sottoclasse e ridefinito), il metodo in questione deve essere stato dichiarato virtuale con la **parola chiave virtual** nella sovraclasse. Chiariamo il tutto con un esempio:
Traduttore Web (translatetheweb.com)

```
#include <iostream>
using namespace std;

class A {

int x=5;

public:
virtual void display() {
cout << "Value of x is : " << x<<endl;
}
};

class B:
```

```
public A {

int y = 10;

public:
void display() {
cout << "Value of y is : " <<y<<endl;
}
};

int main() {

A *a=new B();



a→display();

return 0;
}
```
Nota1: dato che ho dichiarato virtual il metodo display() della sopraclasse A, il compilatore di c++ saprà che l'invocazione nella funzione main() del metodo display() è riferita all'oggetto istanziato dalla sottoclasse B (in java, tutti i metodi non statici sono per default virtuali).

Nota2: il concetto di metodo virtuale non funziona con metodi in overloading.

Classi astratte.

Nota1: un metodo può essere anche denominato funzione membro.
Nota2: una funzione virtuale pura è una funzione che ha il suo prototipo impostato a zero: virtual tipo nomefunzione(lista parametri)=0;
Il solo scopo di una funzione virtuale è quello di avvisare il compilatore che non vuole essere definita.

Una **classe astratte** è una classe che ha dichiarato una o più funzioni virtuali pure. Per una classe astratta non possono essere istanziati oggetti. Per una classe astratta è possibile derivare altre classi, che ereditano le funzioni membro della classe astratta. Le classi derivate devono definire le funzioni membro che hanno ereditato. Una classe astratta ha lo scopo di fornire un'interfaccia di tipo generico. Inoltre una classe astratta permette

di definire una gerarchia di classi. Possiamo ulteriormente aggiungere che nulla vieta ad una classe astratta di definire funzioni membro e dati membro completamente definiti. Esempio:
C++ con simpatia (libero.it)

```
class Figura
{
protected double lato;
  public:
  Figura(double lato)
{
this- >lato=lato;
}
  virtual double area()=0;
};

class TriangoloEquilatero:
public Figura
{
  private:
double altezza;

public:
TriancoloEquilatero(double lato, double altezza):Figura(lato)
{
this- >altezza=altezza;
}
double area(double altezza, double lato)
{
double area;
return area= (altezza*lato)/2;
}
};
```

Nota1: Il costruttore non costruisce un oggetto, è usato per inizializzare un oggetto.
Nota2: una classe astratta ha sempre un costruttore. Il costruttore della classe astratta inizializza gli attributi della classe derivata. Se nella classe astratta non è definito un costruttore (questo significa che la classe astratta non ha attributi), il compilatore assegnerà un costruttore predefinito alla classe astratta.
Nota3: il costruttore della classe astratta Figura viene invocato tramite tramite il suo nome, per inizializzare l’attributo “lato” ereditato dalla sottoclasse TriangoloEquilatero.
Nota4: tutti i metodi astratti ereditati da una sottoclasse devono essere definiti per forza, per evitare che la sottoclasse a sua volta diventi una classe astratta.
Nota5: un metodo astratto è un metodo che è soltanto dichiarato e non definito.

Le stringhe in c++.

Una stringa in java è una costante, cioè non è possibile modificarla con nessun metodo.
Invece in c++ una stringa non è una costante, quindi è modificabile con un metodo.
La classe string contine alcuni metodi utili alla gestione delle stringhe.
Una stringa in c++ non deve essere dichiarata come puntatore ma come una variabile normale.
Per questo non utilizzo la notazione -> per invocare un metodo della classe stringa, ma utilizzo l'operatore punto.

Esempio: la funzione split() prende come parametro una stringa e un carattere separatore. La funzione split() restituisce una lista di oggetti di tipo string, dove ciascun oggetto rappresenta una parola.

```
    string str = "giuseppe;rossi;30";
    vector<string>* lista = split(str, ';');
    cout << str << endl;
    for (int i = 0; i < lista->size(); ++i)
        cout << lista->at(i) << " ";


}
vector<string>* split(string str, char delim) {
    vector<string>* lista = new vector<string>;
    int pos;
    while ((pos = str.find(delim)) != string::npos)
    {
        lista->push_back(str.substr(0, pos));
        str.erase(0, pos + 1);
    }
    lista->push_back(str);

    return lista;
```

Nota1: nella funzione split(), la lista viene allocata in modo dinamico utilizzando l'operatore new. Inoltre non viene perso il riferimento alla lista una volta che la funzione split() termina, poiché la stessa funzione restituisce il riferimento alla lista.

Nota2: in c++ (al contrario di java), un oggetto allocato in modo dinamico non cessa di esistere anche se viene cancellato ogni suo riferimento.

Nota3: in c++, un riferimento ad un oggetto viene memorizzato in un puntatore.
Nota4: quando definisco una funzione il suo prototipo deve essere riscritto fuori dalla funzione main().
Nota5: il metodo find() della classe string cerca e restituisce la prima posizione del carattere, che gli è stato passato come parametro, della stringa che ha invocato il metodo.
Nota6: la variabile di classe npos contiene la risposta negativa del metodo find().
Nota7: in c++ l'accesso diretto ad una variabile di classe è possibile tramite l'operatore di scope ::
Nota8: in c++ anche l'invocazione di un metodo è possibile tramite l'operatore di scope ::
Nota9: il metodo split() viene chiamato per valore. Quindi il metodo slip() riceverà delle copie.
Nota10: dato che l'oggetto array dinamico è associato ad un puntatore, i suoi metodi gli invoco con la notazione - >.
Nota11: in c++, per inserire un valore oppure un oggetto alla fine di un array dinamico (l'array dinamico è un oggetto della classe vector), viene usato il metodo push_back().
Nota12: in c++, tutti gli oggetti (tranne quelli di tipo string) vengono associati a puntatori.
Nota13: il metodo substr() crea e restituisce una sotto stringa relativa alla stringa che ha invocato il metodo. Infatti il metodo substr() ha come primo parametro la posizione del primo carattere della stringa da copiare nella sottostringa, e come secondo parametro il numero di caratteri da copiare nella sottostringa.
Nota14: per evitare che ad ogni iterazione il metodo push_back() inserisca sempre lo stesso elemento nell'array dinamico "lista", devo usare il metodo erase() della classe string per cancellare la parte della stringa che ho già usato per creare la sottostringa da inserire nell'array "lista".
Nota15: il metodo erase() permette di cancellare parte della stringa che ha invocato il metodo. Il primo parametro del metodo erase() mi indica il primo carattere da cancellare nella stringa. Il secondo parametro del metodo erase() indice l'ultimo carattere da cancellare nella stringa.

Nota16: dato che nella stringa “str” tutte le parole (tranne l’ultima) erano separate da “;”, l’ultima verifica della condizione nel while non sarà rispetata e quindi non mi permetterà di inserire l’ultima parola nell’array dinamico “lista”. Quindi devo inserire l’ultima parola esplicitamente con il metodo push_back() senza usare il metodo substr() della classe stringa (tanto alla stringa “str” gli sono state tolte tutte le parola tranne l’ultima).
Nota17: in java non esistono metodi che permettono di modificare una stringa;
Nota18: un array dinamico può utilizzare il metodo at(), che se gli viene passato come parametro la posizione di un oggetto nell’array dinamico stesso, mi restituisce la posizione di quel oggetto.

Flussi di input e output.

In c++, la libreria iostream contiene classi che permettono di rappresentare lo standard input e lo standard output. L’oggetto cin (stream input) permette di gestire il flusso di dati da tastiera. Invece l’oggetto cout (stream output) permette di gestire il flusso di dati verso il video. In particolare l’oggetto cin e l’oggetto cout rappresentano rispettivamente lo standard input e lo standard output.
(vedere lo standard input e lo standard output sugli appunti di fondamenti di programmazione java).
La gestione dei file è affidata alla libreria fstream. In particolare, per la lettura (input) da file utilizzeremo l’oggetto in di classe ifstream (stream di input), e per la scrittura (output) su file utilizzeremo l’oggetto out della classe ofstream (stream di output). Ad esempio, programma che legge un file “tab.txt” e ogni volta che trova un carattere di tabulazione “/t” allora in file “tab1.txt” memorizza 5 caratteri ‘*’ **.** Mentre se non trova il carattere di tabulazione copia il carattere di “tab.txt”. In pratica se il file “tab.txt” è formato da aaa;\tbbb\tddd nel file tab1.txt avremo
aaa;*****bbb*****dddd:

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
ifstream in(“tab.txt”);
ofstream out(“tab1.txt”);
```

```
if(in.is_open()==false || out.is_open()==false)
retun -1; //errore
```

Nota1: l’istruzione ifstream in(“tab.txt”); è caratterizzata dalla dichiarazione dell’oggetto in di tipo ifstream, e dell’invocazione del suo costruttore attraverso le parantesi tonde e passandogli come parametro “tab.txt”. Quindi l’oggetto in non viene costruito in modo dinamico ma in modo statico perché non viene usato l’operatore new. L’istruzione ifstream in(“tab.txt”) crea un canale di comunicazione tra il programma e il file tab.txt, affinché sia possibile fare delle operazioni di lettura sul file.

Nota2: l’istruzione ofstream out (“tab1.txt”); crea un canale di comunicazione tra il programma e il file tab1.txt, affinché sia possibile fare delle operazioni di scrittura sul file.

Nota3: il metodo is_open() è definito sia nella classe ifstream, sia nella classe ofstream e restituisce vero se il canale di comunicazione tra programma e file (passato come parametro) è stabilito. Se il metodo is_open() restituisce falso significa che il file non è stato trovato o non esiste.

Nota4: nella creazione del canale di comunicazione tra programma e file su cui voglio leggere, cioè ifstream in(“tab.txt”), io devo effettivamente creare il file tab.txt, cliccando, in visual basic, con il tasto destro sulla cartella con il nome del progetto e andando sulla voce aggiungi nuovo elemento cerco come tipo di file il file di testo.

Nota5: nella creazione del canale di comunicazione tra programma e file su cui voglio scrivere, cioè ofstream out(“tab1.txt”), il file tab1.txt viene creato in automatico, o se già esiste il file viene sovrascritto.

```
char c;

while (in.get(c)){
if(c==’\t)

for(int=i=0;i<5; i++)
out.put(‘*’);

else
out.put(c);
```

```
}
in.clouse();//chiude il canale di comunicazione tra il programma e il file tab.txt
out.clouse();//chiude il canale di comunicazione tra il programma e il file tab1.txt
```

Nota1: l'invocazione del metodo get() della classe ifstream() ogni volta che viene invocato da in (canale di comunicazione tra programma e file tab.txt) restituisce e memorizza nel parametro attuale c un nuovo carattere presente nel file tab.txt o il valore di fine file (EOF).

Nota2: il carattere di escape ("spazio") viene rappresentato nel codice sorgente con la notazione '\t'. Invece il carattere di escape a video e invisibile.

Nota3: l'invocazione del metodo put() della classe ofstream() ogni volta che vine invocato da out (canale di comunicazione tra programma e file tab1.txt) scrive un nuovo carattere nel file tab1.txt.

Nota4: potevo anche utilizzare l'operatore di flusso >> al posto del metodo get() e l'operatore di flusso << al posto del metodo put(), esempio:

```
string s;
in>>s; // legge dal file tab.txt tutti i caratteri fino a che non trova
       // un carattere di escape '\t' e gli memorizza in s

out<<s; // scrive il contenuto di s nel file tab1.txt
```

Esempio di scrittura di numeri casuali su un file:

```
#include <iostream>
#include <fstream>
#include<math.h>
using namespace std;

int main() {

ofstream out("numeri.txt", ios::app);
srand(time(NULL));

for(int i=0; i<5; i++) {
out<<rand() % 20+1;
out<<" ";
out<<endl;}

out.close();
```

Nota1: Per evitare che ad ogni esecuzione della funzione main() il file numeri.txt venga sovrascritto deve essere usato l’attributo statico app della classe ios. L’attributo statico app “appenderà” alla fine del file numeri.txt i numeri casuali generati da questa esecuzione della funzione main().

Nota2: la funzione srand() inizializza la funzione per la generazione dei numeri casuali (rand()). In altre parole la funzione srand() cambia il punto iniziale (seme) della sequenza di numeri casuali specificata dall’istruzione rand()%20+1;. In particolare il punto iniziale della sequenza di numeri casuali può essere scelto in maniera casuale usando come argomento della funzione srand() la funzione time() impostata a 0 o a null. Se non scrivo esplicitamente la funzione srand() il compilatore la eseguirebbe lo stesso però impostandola a 1 ed ecco perche la funzione rand() restituirebbe sempre lo stesso numero casuale.

Nota3: usiamo l’operatore modulo % con la funzione rand() per restituire un numero casuale specificato in un intervallo : rand()%20 (intervallo di numeri tra 0 e 20). Nell’istruzione rand()%20+5 l’aggiunta di +5 (offset) incrementa di 5 entrambi gli estremi dell’intervallo.

Nota4: l’istruzione out<<” “; scrive nel file numeri.txt uno spazio dopo che è stato scritto un numero casuale (sempre nel file numeri.txt).

Nota5: l’istruzione out<<endl; scrive nel file numeri.txt uno “spazio a capo”.
Numeri random in C++: le funzioni rand() e srand() | MRW.it

Programma che calcola la media dei numeri presi dal file numeri.txt usato nell’esempio precedente:

```
ifstream in(numeri.txt);


int x=0;
int s=0;
int conta=0;

for(;;){
in>>x;
if(in.eof())
break;
```

```
conta++;
s=s+x;
}
cout<<(double)s/conta;
in.close();
```

Nota1: siccome non conosco quanti numeri ci sono nel file numeri.txt, sfrutto il ciclo for senza condizione per leggere ad ogni ciclo il numero corrente nel file numeri.txt. Per ogni ciclo dopo aver memorizzato nella variabile “x “il numero corrente nel file numeri.txt controllo con la funzione eof() se sono arrivato alla fine del file. Se la risposta a questo controllo è positiva significa che ho letto tutti i numeri del file numeri.txt, e utilizzo l’istruzione di break per interrompere il ciclo. Invece se la risposta a questo controllo è negativa , la variabile “conta” viene incrementata di 1 (che poi verrà usata per calcolare la media dei numeri del file numeri.txt) e il numero corrente (memorizzato nella variabile “x”) viene sommato con il risultato della somma del ciclo precedente (memorizzato nella variabile “s”).

Nota2: l’istruzione cout<<(double)s/conta; sfrutta il canale di comunicazione tra il programma e il file numeri.txt, già aperto dall’istruzione ifstream in(“numeri.txt”), per scrivere sul file la media dei numeri che ho letta dal file. Quindi l’oggetto cout (della libreria iostream) con l’operatore di flusso << non serve solo a rappresentare lo standard output(cioè la scrittura su schermo). Inoltre, utilizzo l’operatore di cast (double) per rendere più preciso il risultato della media.

Nota1: concatenazione di stream;
Nota2: ascoltare la lezione del 11-05 -2021 al punto 00-40-30.
Nota3: concetto di stream in c++.